# Automatically Resolving Data Source Dependency Hell in Large Scale Data Science Projects

Laurent Boué
*Microsoft*
Israel
laurent.boue@microsoft.com

Pratap Kunireddy
*Microsoft*
India
pkunireddy@microsoft.com

Pavle Subotić
*Microsoft*
Serbia
pavlesubotic@microsoft.com

*Abstract*—**Dependency hell is a well-known pain point in the development of large software projects and machine learning (ML) code bases are not immune from it. In fact, ML applications suffer from an additional form of dependency hell, namely, *data source dependency hell*. This term refers to the central role played by data and its unique quirks that often lead to unexpected failures of ML models which cannot be explained by code changes. In this paper, we present an automated data source dependency mapping framework that allows MLOps engineers to monitor the whole dependency map of their models in a fast paced engineering environment and thus mitigate ahead of time the consequences of any data source changes. Our system is based on a unified and generic approach, employing techniques from static analysis, from which data sources can be identified on a wide range of source artifacts. Our framework is currently deployed within Microsoft and used by Microsoft MLOps engineers in production.**

*Index Terms*—**data dependency, data science, static analysis**

## I. Introduction

With the widespread adoption of machine learning (ML) models, it is common for large data science organizations to run hundreds of mission-critical workloads daily. Any failures or delays negatively affect stakeholders who do not receive the information they depend on to generate business value and deliver results to their customers. In this context, a new discipline that aims to streamline and standardize all the engineering pieces necessary to land production-level quality ML models at scale has emerged under the name of MLOps. Although similar in spirit and execution to DevOps, MLOps engineers must also deal with a new set of challenges associated with data-centric (instead of code-centric) projects [1], [2]. MLOps includes incident management, data quality processes (model / data drift checks), data contracts, operationalization / support of models and other responsibilities to automate the lifecycle and continuous delivery of high-performing models in production. All this must be accounted for, while ensuring scale, latency, speed and optimal costs.

In particular, the topic of *data source dependencies* is central to MLOps. As models evolve and grow in scope, the overall MLOps architecture rapidly becomes overwhelmed by a new form of *dependency hell* where, instead of software packages' versions, MLOps engineers now must try to resolve models' data source dependencies. Generally, this task is even more laborious because data dependencies tend to span across organizational boundaries and originate from a diversity of data stores. It is essential to note that while code issues surface occasionally, data issues arise frequently and rather unpredictably. Unfortunately, important data source announcements (such as delays or issues of any kind) which are often communicated via distribution list emails commonly go unnoticed by model owners thereby resulting in model failures and ultimately negative customer impact. Even when data teams assign a specific role to manually track all such announcements, the task of deciphering which models are impacted remains daunting due to the inherently iterative nature of data science models and to their complex dependency graphs (with transitive and model-model structures).

In this paper, we present an automated dependency mapping framework to extract data source dependencies from ML model workloads. It allows subscribed systems to automatically notify a model owner if, for example, a dependent table schema has changed, contains corrupted data or any other type of modification that is likely to affect the ML model. Upon receipt of the notification, the model owner can quickly take the necessary mitigating actions (e.g., re-train model, ignore data, change data source etc.) so that model consumers are not affected. To adequately perform automated data source mapping, our technique employs static analysis on a set of connected code artifacts (activities) referred to as *activity graphs*. The advantage of our approach is that we can compute data source dependencies quickly and with ease thus avoiding run-time complications (e.g., security configurations, storage costs etc.). Moreover, our framework is extensible in that it supports a wide range of activity artifacts including queries, scripts and notebooks without requiring modifications. We have implemented our automated data source dependency mapping framework and exposed it as a web API service. Our implementation is deployed within Microsoft Cloud Data Sciences (MCDS) where it is used to mitigate and prevent data source related incidents ahead of time. We summarize our contributions as follows:

- We present a novel dependency mapping framework that extracts data sources from activity graphs using static analysis.
- We present an implementation of our technique and deployment in real-world industrial use cases.

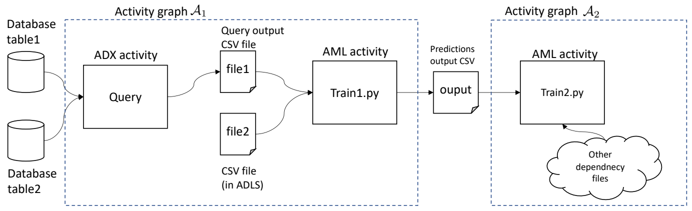

Fig. 1: Dependency Mapping Example

- We evaluate our framework and show its utility on real-world benchmarks.

## II. Problem Definition

The need to automate dependency mappings comes from the typical volumes as well as time scales associated with data sources. For instance, a modestly-sized data science organization deployed on our framework needs to manage approx. 300 activities, 25 databases, 250 tables and 2,500 columns. Additionally, models are continuously updated and modified by model owners so that data sources routinely appear or disappear on an hourly basis. On top of this, models and data sources often display complex transitive dependencies with one another. In this context, it is infeasible to manually track data source dependencies. Automatic monitoring also allows data science organizations to reduce their rates of missed Service Level Agreements (SLAs) with their stakeholders by providing a faster path to identify root cause reasons for model failures and resolve incidents before they are even noticed by downstream stakeholders.

A typical ML workload is constructed as one or several *activity graphs*. An activity graph is a set of Azure Data Factory (ADF) *activities* such as Azure Data Explorer (ADX) or Azure Machine Learning (AML). Activities digest data and produce output data. Hence, activities are connected via intermediate data. We classify data into two categories, namely, initial data sources (i.e., raw data from a database or file etc.) and derived data source (i.e., output from another activity).

Activities perform a well-defined unit of work such as control flow operations, data mapping operations and many others using programming logic i.e., scripts, database queries etc. Normally, ADX activities are responsible for data querying and AML activities for training / inference of ML models. We define a data source as a tuple $\langle s, \bar{c} \rangle$ where $s$ is a source symbol i.e., representing a database table name or filename, $\bar{c}$ is the set of columns. We use the notation $s^{\bar{c}}$. Typically, an activity graph is defined *per* model and its output is the output of a model e.g., inference. Output from a model i.e., an activity graph, can be an input into another activity graph.

*Problem 2.1 (Model-Input Dependency Mapping):* Given a set of initial data sources and a set of models in activity graphs, determine which initial data sources impact which models.

*Example 2.1 (Motivating Example):* In Figure 1 we describe a small configuration with two activity graphs $\mathcal{A}_1$ and $\mathcal{A}_2$ (enclosed by the dotted lines). In the first graph, we have two activities, namely, an ADX database query and an AML Python script. The database query is performed on two database tables and on some set of columns (which we don't define for readability). Thus *table1* and *table2* are initial data sources whereas *file1* is a data source derived from the ADX activity. Both *file1* (derived data source) along with another *file2* (initial data source) serve as inputs to the AML activity consisting of a machine learning script *Train1.py* which stores its predictions in an *output* file. To slightly add to the complexity, the derived data source *output* which is produced by $\mathcal{A}_1$ is used as input to train another model *Train2.py* in another AML activity which belongs to the second activity graph $\mathcal{A}_2$.

## III. Dependency Mapping Framework

In this section we describe our dependency mapping algorithm. Our algorithm operates on three levels: (1) set of connected activity graphs (2) a single activity graph (3) a single activity.

**Algorithm 1:** Activity Graph Analysis

```
   Input: Activity graph $\mathcal{A}$
   Result: Set of data sources $\zeta$
1  $\zeta := \emptyset$;
2  $nodes$ := FIFO();
3  $nodes$.push($\mathcal{A}$.start);
4  while $nodes$ is not empty do
5      a := $nodes$.pop();
6      I := Analyze(a);
7      if fixpoint reached then
8          return $\zeta$;
9      end
10     for $i \in I \wedge \mathcal{A}.derived(i)$ do
11         nodes.push($\mathcal{A}.deps(i)$);
12     end
13     $\zeta := \zeta \cup \{i \in I \mid source(i)\}$;
14 end
15 return $\zeta$;
```

Algorithm 1 describes our approach for computing a set of data source dependencies $\zeta$ for a single activity graph $\mathcal{A}$. For each $\mathcal{A}$ we assume a start activity $start$, a function $deps$ that given a data source provides a set of activities in $\mathcal{A}$ that produced it as output and a predicate $derived$ which asserts if a data source is derived. In line 1, we initialize the data source set $\zeta$ to the empty set. We then create a FIFO ($nodes$) and add the starting activity to it. The general idea is that we propagate backwards (line 11) to other connected activities until we encounter only initial data sources or stop gaining information (reach a fixpoint). Each time we encounter an activity, we analyze the activity statically using the **Analyze** function (line 6). We then proceed to follow dependencies (via $deps$) from derived data (line 10) and update $\zeta$ when initial data sources (line 13) are found. The algorithm terminates when either no more nodes are in the FIFO (line 15) or we detect a fixpoint (due to the assumed monotonicity of **Analyze**) in line 8.

```
...
...
data1 = pd.read_csv("file1.csv")
data2 = pd.read_csv("file2.csv")
X = data1[["loc", "age"]]
y = data2[["target"]]
X_train, X_test, y_train, y_test =
    train_test_split(X, y, ...)
lr = LogisticRegression()
a = lr.fit(X_train, y_train)
y_pred = lr.predict(X_test)
y_pred.to_csv("output.csv")
```

Fig. 2: Train1.py Script

*Example 3.1 (Motivating Example Cont.):* Consider our motivating example in Figure 1. Suppose we analyze the first activity graph. We first process *Train1.py* and have $\zeta = \{file2^{age}\}$ and mark $file1$ as a derived data source. We then add the query activity to our *nodes* and find as a result that we have $\zeta = \{file2^{age}, table1^{loc}, table2^{name}\}$ assuming the query selects columns $loc$ and $name$.

For the case of several activity graphs we apply Algorithm 1 to individual activity graphs and repeatedly make the following inference: $\dfrac{o \in O_A \quad o \in \zeta_B}{\zeta_B = (\zeta_B - o) \cup \zeta_A}$ until a fixpoint is reached. Here, $O_A$ is the set of all outputs of the a model in an activity graph $A$, and $\zeta_B$ is a mapping in an activity graph $B$. This rule states: *if there is a common element between an output of activity graph A and a data source mapping of activity graph B then there is a transitive mapping to activity graph B for all data sources that are mapped to activity graph A, excluding the common element.*

*Example 3.2 (Motivating Example Cont.):* Consider our motivating example in Figure 1. We first build the graph for the model in *Train2.py*. Here $\zeta_{\mathcal{A}_2} = \{output^{\cdots}, file2^{name}, \dots\}$. Since $O_{\mathcal{A}_1} = \{output^{\cdots}\}$ and $\zeta_{\mathcal{A}_1} = \{file2^{age}, table1^{loc}, table2^{name}\}$ we can then conclude that $\zeta_{\mathcal{A}_2} = \{file2^{age}, table1^{loc}, table2^{name}, \dots\}$.

## IV. Dependencies From Source Code

In this section we describe our static analysis techniques to implement the **Analyze** function in line 6 of Algorithm 1.

### A. Source Mapping Analysis for Scripts

*a) Static Analysis:* To statically analyze scripts we perform an abstract interpretation. The framework of abstract interpretation [3], [4], computes an over-approximation $\sigma^\sharp$ of a state $\sigma$ by iteratively solving the fixpoint equation $\sigma^\sharp = \sigma_0^\sharp \sqcup [\![p]\!]^\sharp(\sigma^\sharp)$ using a monotonic interpretation of the abstract semantics $[\![p]\!]^\sharp$ for a program $p$, that produces an updated abstract state that is joined ($\sqcup$) with the initial abstract state $\sigma_0^\sharp$. We refer the reader to [5], [6] for an in-depth explanation of abstract interpretation and data flow analysis.

Practically, the above is achieved by converting source code into an abstract syntax tree (AST) representation and subsequently into a control-flow graph (CFG), which is then analyzed. Given a sequence of statements, the CFG is a directed graph that encodes the control flow of the nodes that represent straight-line statements (no branching etc.). We define a CFG as $\langle L, E\rangle$ where an edge $(l, st, l') \in E$ reflects the semantics of statement $st$ associated with the CFG edge from locations $l$ to $l'$. The set of variables in all the statements is denoted by $V$ and the set of symbols by $S$. The analysis proceeds to compute a (least) fixpoint solution by processing each statement starting with the entry statement and following the CFG control-flow using a worklist algorithm [6].

To instantiate an abstract interpretation for our dependency mapping problem, we need to define two elements (1) how we represent that abstract computational state and (2) how to define the abstract semantics i.e., rules for how we process each type of statement.

*b) Abstract state:* We define an abstract state as a mapping between variables and set of data sources e.g., $\sigma^\sharp = v \mapsto \{\dots, s^{\bar{c}}, \dots\}$ where $v \in V$, $s \in S$ is a data source symbol with selected set of columns $\bar{c}$. We define a $\sqcup$ operator on states as a piece wise set union. We denote substitution of a variable $x$ with value $d \subseteq S$ in an abstract state $\sigma^\sharp$ as $\sigma^\sharp[x \mapsto d]$. Apart from the abstract state, we also introduce a *mapping set* $I$ which contains the data sources that reach a model function.

*c) Abstract semantics:* To define our abstract semantics we introduce an abstract transformer, namely, a function $[\![st]\!]$ parameterized by a statement type *st* that takes as arguments (denoted as $\lambda$ arguments) an abstract state $\sigma^\sharp$ and mapping set $I$. The abstract transformer specifies how we analyze each type of statement and returns an updated abstract state and mapping set. Moreover, our abstract semantics requires us to detect input and model operation statements. We thus mark the set of input statements (e.g., read_csv) in a set *Source* and the set of model training statements (e.g., fit) in a set *Sink*. The result of an analysis is a mapping set $I$ which is a set of data sources. Note, by the definition of our problem, we assume only one model per activity graph.

Below, we outline the key set of rules that govern how input sources are propagated for a given statement.

1) **input:**
$$\lambda\sigma^\sharp.I.[\![y = \text{read}(input)]\!] = \langle \sigma^\sharp \cup y \mapsto \{input^C\}\rangle, I$$
where read $\in$ *Source*, $C$ set of all columns of input

2) **project:**
$$\lambda\sigma^\sharp.I.[\![y = x.sel[\bar{c}]]\!] = \sigma^\sharp[y \mapsto \sigma^\sharp(x)^{\bar{c}}], I$$
where $\sigma^\sharp(x)^{\bar{c}}$ constrains columns of all source mapped to $x$ to columns $\bar{c}$ where applicable

3) **external functions:**
$$\lambda\sigma^\sharp.I.[\![y = f(\bar{x})]\!] = \sigma^\sharp[y \mapsto \sigma^\sharp(y) \sqcup \bigsqcup_{x \in \bar{x}} \sigma^\sharp(x)], I$$

4) **sink**:
$$\lambda\sigma^\sharp.I.[\![m.f(\bar{x})]\!] = \sigma^\sharp, \forall x \in \bar{x}, I \cup \sigma^\sharp(x)$$
where $f \in$ *Sink*

Case (1) handles read statements. Here we simply map the left-hand-side variable to the source being read with all columns. Case (2) handles projection. Here we map the left-hand-side variable to the source with appropriate constraints on columns. Case (3) handles external functions. Here we simply join all sources from the arguments ($\bar{x}$) of the function to the left-hand-side variable. Case (4) handles a call to a model function. Here we add the sources of the arguments to $I$ for all inputs arguments to the function.

*Example 4.1 (Static Analysis Example):* For *Train1.py* in Figure 2, we proceed to process rule (1) for both read statements, finishing with the abstract state: $\{data1 \mapsto file1^{\dots,loc,age,\dots}, file2 \mapsto output^{\dots,target,\dots}\}$.

The projection invokes rule (2) constraint so that we have variables $X$ and $y$ map to: $\{X \mapsto file1^{loc,age}, y \mapsto output^{target}\}$

Finally, at line 8, we apply rule (3) and when we detect the fit function in line 10 we apply rule (4) and add the data sources of $X\_train$ and $y\_train$ to $I$, i.e., $I = \{file1^{loc,age}, output^{target}\}$.

### *B. Source Mapping Analysis for Other Artifacts*

Aside from scripts, other activities we support include database queries, multi-file scripts and notebooks. For example, queries can be converted to imperative Intermediate Representations (IRs) (cf. Chapter 3 in [7]) and subsequent CFGs analyzed like scripts in Section IV-A. Multi-file and function scripts can be resolved via cloning [8]. Notebooks can be handled by the technique in [9].

## V. Deployment and Applications

In this section we describe the implementation of our mapping framework and its deployment. We also discuss several use cases.

The high-level overview of the architecture of the mapping framework API is presented in Figure 3. We have deployed the dependency map as a REST web API service. The request payload of the API consists of: (1) the path to an Azure DevOps Git repository hosting the target source code we wish to analyze and extract data dependencies from (2) An optional field to filter the response. The response consists of the dependencies of the models in the Git repository, filtered as per the optional field in the request. The overall pipeline is orchestrated within Azure Data Factory and can be scheduled to run either at a pre-defined schedule or at event based. The code itself is containerized and the image contains the .NET runtime. First, the target source code is pulled from Azure DevOps Git repository and processed by our mapping framework. Finally, a copy of the data source dependency mappings is stored as a file in Azure Data Lake Storage (ADLS). The service controllers are hosted separately on a Kubernetes cluster provided through Azure Machine Learning Inference Endpoints and the service pods cache the output file and serve the real-time requests. Below we describe three use cases that use our service.

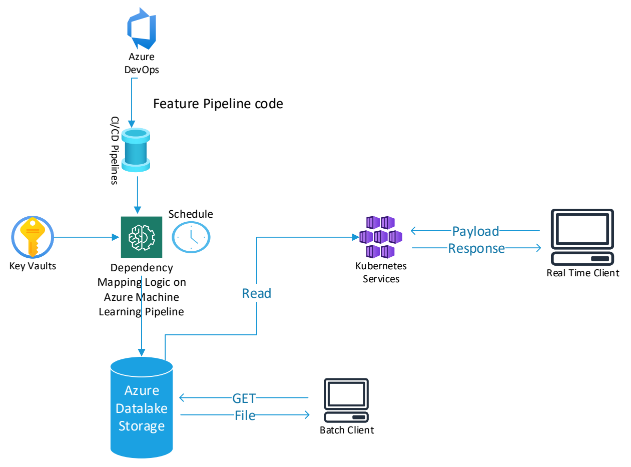

Fig. 3: High-level sketch of the software architecture implemented to expose dependency map as a RESP web API.

*a)* ***Use Case I:*** *Interactive MLOps dashboards:* Our first use case is an interactive PowerBI dashboard which hooks into the API for real-time updates. The main purpose of the dashboard is to assist our MLOps engineers in maintaining good overall health of our production ML models. For example, when MLOps engineers receive communication about data source changes, they rely on the dashboard to identify which activities may be impacted. This way, we ensure that comprehensive and responsive actions are taken to react to data source changes. In particular, the dashboard is useful in identifying non-trivial dependencies such as model-to-model dependencies. Finally, the dashboard helps MLOps engineers decide if a model needs re-training in case some of its most important features are affected by a data source change.

*b)* ***Use Case II:*** *Knowledge graphs:* Traditional entity relationship diagrams quickly become outdated with ever changing data sources, model owners and downstream business stakeholders. Recently, knowledge graphs have gained momentum as a way to collect information about complex datasets, their relationships to one another, connect business terms to data elements and more. Additionally, knowledge graphs are a convenient tool to surface insights that improve data discovery and governance. We have implemented our own version of a knowledge graph that leverages the dependency map API to cross-link between activities and their extended information. Integrating this network of information with our Incident Management System (IcM) removes some of the manual load from MLOps engineers by sending automatic alerts to the relevant parties affected by a data source change. Additionally, the dependency map API populates meta-information knowledge about the data sources to improve discoverability (such as column data types in tables, business stakeholders etc.).

*c)* ***Use Case III:*** *Feature stores:* Feature stores are a data management layer whose purpose is to act as a single source of truth of ML datasets. Persisting features in a principled way is expected to accelerate model training and scoring so

TABLE I: Dependency Map Evaluation

| Model ID | Avg. Act Size | No. Act | No. Dep | $\|\zeta\|$ | T (ms) |
|---|---|---|---|---|---|
| M1 | 1928.9 | 63 | 127 | 1591 | 742 |
| M2 | 199 | 1 | 37 | 1209 | 336 |
| M3 | 4575.2 | 38 | 37 | 1145 | 986 |
| M4 | 2526.8 | 18 | 47 | 978 | 302 |
| M5 | 2500.7 | 12 | 31 | 446 | 126 |
| M6 | 1520.3 | 13 | 60 | 434 | 233 |
| M7 | 2808.8 | 8 | 20 | 282 | 82 |
| M8 | 713.2 | 6 | 59 | 206 | 72 |
| M9 | 640.1 | 7 | 55 | 199 | 80 |
| M10 | 1428 | 5 | 26 | 170 | 122 |
| M11 | 2171.3 | 6 | 35 | 135 | 164 |
| M12 | 3571 | 11 | 44 | 131 | 162 |
| M13 | 92.2 | 18 | 34 | 100 | 15 |
| M14 | 1244 | 18 | 111 | 58 | 599 |
| M15 | 2267.4 | 8 | 55 | 39 | 163 |
| M16 | 927.5 | 2 | 22 | 32 | 87 |
| M17 | 569 | 2 | 47 | 29 | 24 |
| M18 | 84 | 1 | 39 | 27 | 22 |
| M19 | 2001 | 1 | 25 | 24 | 53 |
| M20 | 1239.8 | 4 | 42 | 22 | 81 |
| M21 | 1305.5 | 2 | 52 | 21 | 76 |
| M22 | 7584.7 | 3 | 63 | 21 | 136 |
| M23 | 682 | 1 | 23 | 20 | 49 |
| M24 | 2392.5 | 8 | 40 | 20 | 134 |
| M25 | 1615.7 | 3 | 26 | 19 | 83 |
| M26 | 798 | 3 | 31 | 18 | 59 |
| M27 | 743.7 | 6 | 43 | 4 | 719 |
| M28 | 748.5 | 2 | 42 | 4 | 177 |
| M29 | 142 | 1 | 35 | 3 | 45 |
| M30 | 84 | 2 | 65 | 2 | 8 |
| M31 | 373.5 | 4 | 25 | 2 | 70 |
| **Geo. Mean** | 967.1 | 4.8 | 40.8 | 50.3 | 107.8 |

that compute-heavy features need not be re-evaluated every time they are requested. Our dependency map API plugs into a feature store to identify feature re-usability and decrease the storage requirements even more. For example, it is very common for multiple graphs to share a number of data sources. This happens when different models re-use the same set of features. The dependency map API automatically populates the feature store meta-information so that data source duplication is explicitly taken into account in the design of the feature store persistence model.

## VI. Experimental Evaluation

In this section, we evaluate our implementation of our data source dependency mapping framework.

*a) Experimental Setup:* We perform our evaluation based on 31 real-world ML models that are leveraged by downstream stakeholder teams which rely on the output of these models to make more informed business decisions. 20% of models require an inter-graph analysis. Our experimental evaluation aims to investigate the latency of our system and the number of dependencies computed for various model characteristics. All experiments were performed on an Azure Machine Learning workspace with 26 GB RAM running the Ubuntu 22.04.1 LTS operating system.

*b) Experimental Results:* We present our experiments in Table I. Column **Avg. Act Size** measures the average size of each activity by number of tokens in the activity code. **No.Act** is the number of activities in the activity graphs, **No.Dep** is the number of dependencies (i.e., edges) in the activity graphs, $|\zeta|$ is the number of (transitive and initial) data sources found to map to the model, **T** is the execution time (ms) to compute $\zeta$. We summarize each column by providing the geometric mean.

Our experimental results show that all of our dependency mappings can be built in in under a second with a max. 986 ms and with a geo. mean of 107.8 ms. This conforms to execution times for similar static analyses on Python data science scripts [10], [9], [11]. We also observe that the size of the activities, the number of activities and the number of dependencies all have an influence on the execution time, while the size of $\zeta$ doesn't appear to have an influence. This conforms with our expectations as we statically analyse activity graphs and thus are unaffected by number of initial data sources. Overall, our system is able to compute mappings after each pull request is merged so that dependencies are continuously kept up-to-date with respect to the SLAs we typically encounter.

## VII. Related Work

The management [12] and development [2] of ML systems is a rapidly emerging research area. Amongst the body of works in this area, our technique most resembles methods that extract information from models using provenance/lineage. For instance, the techniques in [13], [14], [15], [16] provides run-time based lineage of ML pipelines. In contrast to these systems, our system focuses on quickly mapping dependencies between data sources and models by leveraging static analysis. Since we do not require a fine-grain data (actual values) run-time diagnostics provide little benefits and results in unnecessary burdens such as execution logging.

We are unaware of other work that performs static analysis on activity graphs or similar structures. In terms of techniques that leverage static analysis more broadly for ML [9], [11], our technique has similarities with Vamsa [10]. Vamsa builds a static provenance directed acyclic graph (DAG) from a single Python script using a forwards/backwards propagation on acyclic control-flow programs. Compared to Vamsa, we analyze activity graphs which may contain various connected code artifacts including scripts and queries. Our static analysis also has foundational similarities with the dependency analysis in [17], which could be used as an intermediate semantics to prove theoretical soundness.

## VIII. Conclusion

We have presented a dependency mapping framework for large scale ML models that are defined by inter-connected activity graphs. Our technique statically analyzes these activity graphs to compute data sources that may impact their models. We have deployed our framework within MCDS as a service that is leveraged in a number of use cases. In the future, we plan on expanding our coverage of ADF activities.

**Acknowledgments.** We thank all our colleagues at Microsoft. In particular, we thank Kiran R, Kirk Li, Swarnim Narayan, Rituparna Praharaj and Guillaume Boué for their valuable comments.

## REFERENCES

[1] G. Symeonidis, E. Nerantzis, A. Kazakis, and G. A. Papakostas, “Mlops - definitions, tools and challenges,” in *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*, 2022, pp. 0453–0460.

[2] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, “Software engineering for machine learning: A case study,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 291–300.

[3] P. Cousot and R. Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints,” in *Proc. POPL*, January 1977, pp. 238–252.

[4] G. A. Kildall, “A unified approach to global program optimization,” in *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, P. C. Fischer and J. D. Ullman, Eds. ACM Press, 1973, pp. 194–206.

[5] P. Cousot, *Principles of Abstract Interpretation*. MIT Press, 2021. [Online]. Available: https://books.google.rs/books?id=Cwk_EAAAQBAJ

[6] U. Khedker, A. Sanyal, and B. Sathe, *Data Flow Analysis: Theory and Practice*. CRC Press, 2017.

[7] P. Subotić, “Scalable logic defined static analysis,” Ph.D. dissertation, University College London, 2018.

[8] K. D. Cooper, M. W. Hall, and K. Kennedy, “Procedure cloning,” *Proceedings of the 1992 International Conference on Computer Languages*, pp. 96–105, 1992.

[9] P. Subotic, L. Milikic, and M. Stojic, “A static analysis framework for data science notebooks,” in *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022*. IEEE, 2022, pp. 13–22. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP55303.2022.9794067

[10] M. H. Namaki, A. Floratou, F. Psallidas, S. Krishnan, A. Agrawal, Y. Wu, Y. Zhu, and M. Weimer, “Vamsa: Automated provenance tracking in data science scripts,” in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1542–1551.

[11] P. Subotic, U. Bojanic, and M. Stojic, “Statically detecting data leakages in data science code,” in *SOAP ’22: 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, San Diego, CA, USA, 14 June 2022*, L. Gonnord and L. Titolo, Eds. ACM, 2022, pp. 16–22. [Online]. Available: https://doi.org/10.1145/3520313.3534657

[12] S. Schelter, F. Biessmann, T. Januschowski, D. Salinas, S. Seufert, and G. Szarvas, “On challenges in machine learning model management,” *IEEE Data Eng. Bull.*, vol. 41, pp. 5–15, 2018.

[13] S. Grafberger, S. Guha, J. Stoyanovich, and S. Schelter, “Mlinspect: A data distribution debugger for machine learning pipelines,” in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2736–2739.

[14] X. Xu, C. Wang, Z. Wang, Q. Lu, and L. Zhu, “Dependency tracking for risk mitigation in machine learning (ml) systems,” in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022, pp. 145–146.

[15] M. Vartak, J. M. F. da Trindade, S. Madden, and M. Zaharia, “Mistique: A system to store and query model intermediates for model diagnosis,” in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1285–1300. [Online]. Available: https://doi.org/10.1145/3183713.3196934

[16] M. Vartak, H. Subramanyam, W. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia, “Modeldb: a system for machine learning model management,” in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, C. Binnig, A. D. Fekete, and A. Nandi, Eds. ACM, 2016, p. 14. [Online]. Available: https://doi.org/10.1145/2939502.2939516

[17] F. Drobnjaković, P. Subotić, and C. Urban, “Abstract interpretation-based data leakage static analysis,” 2022.